DPL

DESIGN PATTERNS LIBRARY

PROGRAMMING GUIDE

Rev. 0.50

Design patterns library

Programming guide

Copyright (c) 2010 Samsung Electronics, Inc.

All rights reserved

Author: Przemyslaw Dobrowolski

Contact: p.dobrowolsk@samsung.com

TABLE OF CONTENTS

Installation ... 6

Basic concepts ... 6

Application ... 6

MVC ... 8

- Event ... 8
- Event support ... 8
- Generic events ... 9
- Models ... 10
- Controllers ... 11
- Context inheritance ... 12
- Advanced topics ... 12
  - Manual switch of thread inherited by event support ... 12
  - Generic Event Call ... 13
  - Internal representation of event ... 13

Thread support ... 13

- Thread ... 13
- Advanced topics ... 14
  - Waitable handle watch support ... 14
  - Waitable input output execution context support ... 14

Abstract IO ... 14

- Waitable event ... 14
- Abstract input/output ... 15
- Abstract waitable input/output ... 16
- Abstract socket ... 17
  - Address ... 18
- Generic socket ... 18
- Unix/TCP socket ... 19

Synchronization ... 21

© Samsung Electronics Co., **Confidential**

Mutex 21
Recursive mutex 21
Spin lock 22
Read Write Mutex 22
Remote Procedure Calls 23
Abstract RPC connection 23
RPC Call 23
Generic socket RPC connection 24
UNIX/TCP socket RPC connection 24
Abstract RPC connector 24
Generic socket RPC client/server 25
UNIX/TCP socket RPC client/server 25
Logging system 31
DLOG provider 32
Old style provider 33
Exception system 34
Cryptography 35
Hash functions 35
Algorithms 35
Utilities 36
Atomic 36
SQL Connection 36
Executing simple commands 37
Executing command with result 37
Singleton 38
Noncopyable 39
RAII 39
Scoped Pointer 39
Scoped Array 40

© Samsung Electronics Co., **Confidential**

Shared Pointer 40
Shared Array 41
Scoped Free 41
Scoped Close 41
Single Instance 42
Task / TaskList 42
Binary Queue 42
Asynchronous Semaphore 44
Sample applications 45
RPC Metronome application 45
Synchronous event calls 51

© Samsung Electronics Co., **Confidential**

INSTALLATION

DPL library is provided on top of two basic frameworks: EFL(ECORE) and GTK(GLIB). These two are compiled separately into distinct libraries. Along with two libraries there are two separate *include* directories and two *pkgconfig* files. Following table summarizes debian packages available through apt:

Mode	Binary package	Developer package	pkgconfig file	Debug package
EFL(ECORE)	dpl-efl	dpl-efl-dev	dpl-efl.pc	dpl-dbg
GTK(GLIB)	dpl-gtk	dpl-gtk-dev	dpl-gtk.pc	dpl-dbg

For example: to develop with EFL only packages, we need to install only dpl-efl and dpl-efl-dev packages. To develop application that can be compile-time configured to use EFL or GTK, we must install both DPL packages for EFL and GTK.

DPL support both i386 and ARM platforms.

Basically, there is no difference between GTK and EFL version. By using different *pkgconfig* files, include directories and link libraries are automatically switched to use proper subsystem.

All DPL include files reside under directory <dpl/*>. Examples are:

Example:

```
#include <dpl/application.h>
#include <dpl/geneneric_event.h>
```

Notice that, although application component implementation differs between EFL and GTK version, we use the same include file.

BASIC CONCEPTS

DPL is written using general programming technique. In practice it means that each component uses only concept objects that have certain features. This kind of approach gives great flexibility.

Internal components of DPL are heavily using templates, but it is usually not needed to completely understand internal structure to use all features available in DPL. Library developers need to get know about some advanced programming techniques and guidelines.

All DPL components are placed in namespace DPL.

APPLICATION

As usual we will start with an example “hello world” application. Provided example will run full screen black window.

© Samsung Electronics Co., Confidential

Example:

```
#include <dpl/application.h>

int main(int argc, char *argv[])
{
    DPL::Application app(argc, argv, "hello_world");
    return app.Exec();
}
```

Application class provides several virtual methods that can be overloaded:

Method declaration	Purpose
virtual void OnCreate()	Called when first view of application is created and initialized
virtual void OnStart()	Called at application startup, used to speed up application launch and to display loading banner or message
virtual void OnStop()	Application is being sent to background in multi application environment
virtual void OnResume()	Application is being sent to front in multi application environment
virtual void OnRelaunch()	Application receives 'launch service' with arguments through AUL
virtual void OnTerminate()	Application is requested to free all resources before application exit
virtual void OnLowMemory()	System memory is low and application should free all unused memory
virtual void OnLowBattery()	Application is requested to free all unused memory and prepare to quit
virtual void OnLanguageChanged()	System language has changed and application may reload all language dependant resources

By default all these method do nothing. Any number of them can be overloaded and handled.

To quit an application, call *Quit()* in any of event handlers. A special quit message will be sent and an application and it will quit in next event loop iteration.

© Samsung Electronics Co., **Confidential**

By default, an application will create default full-screen black windows. To override this setting, use non-default parameters in *Application* constructor.

MVC

Model-View-Controller support provided by DPL is an architecture that enables easy implementation of modern applications. To get idea of whole architecture, few basic concepts have to be introduced.

Fundamental idea of MVC is to distinguish and divide application parts such as: logic, data representation (model) and event processing (controllers). These can be defined in various ways, but common idea is similar. Apart from listed components in MVC applications there are many other such as DAO, TO and other. Some of them will be discussed in this paragraph, but to get all of descriptions, reader is encouraged to read some book about MVC paradigm GUI programming.

Most important concept is an event. An event is an abstract message that is sent between various contexts. Events can be generated as a response to other events; they can be spontaneous or sent from framework or environment. All asynchronous communication is done with events. On top of a basic event, there are defined model properties, controllers, asynchronous IO and more.

Second most important concept is execution context inheritance. This basically means that some objects can be gratified a thread that they are allowed to execute their code on. These are mainly controllers and upper-level IO objects (sockets).

EVENT

An event can be a type that supports copying and empty construction. Examples are: int, float, struct, class, and DECLARE_GENERIC_EVENT. Most convenient and suggested way of defining new event is by using macro DECLARE_GENERIC_EVENT.

Remember:

*Events are always copied **by value**. This implies that during copy **each field is copied by value**. Defining an event with **pointer fields** is possible, but to access the memory pointed by event developer must ensure that memory is available for whole the time that event is being processed (from posting it to receiving). The moment of receiving an event **is not known** at the moment of posting it.*

For heavy events that should not be copied, one can use some smart pointer mechanism. For example DPL's *SharedPtr* implementation is sufficient.

EVENT SUPPORT

Fundamental concept of DPL's support for MVC is Event Support. An object that inherits EventSupport template is given a capability of sending and receiving specific type of events.

Consider following trivial example. We would like to have an object representing an integer value. Basically it is an *int* type. We would like to give it an ability to generate events about its changes. What we have to do is to wrap-up simple integer type into a class, and derive proper event support.

At this moment we have to decide what type of event should generate such class. The most obvious selection is an event consisting of its new value. Optionally, we can send delta value or event a pair of old and new values.

We will select first option. Example code is as follows:

© Samsung Electronics Co., Confidential

Example:

```
#include <dpl/event_support.h>

class Integer : public EventSupport<int>
{
    int m_value;

public:
    Integer()
        : m_value(0)
    {
    }

    int Get() const
    {
        return m_value;
    }

    void Set(int value)
    {
        m_value = value;
        PostEvent(m_value);
    }
};
```

An integer value is wrapped by class Integer. To access value, there are two methds: *Ge()t* and *Set()*. *Get()* method is trivial. More interesting is second methods. To set a value, first its value is saved and secondly an event with current value is sent. If there is no synchronization of access to Integer, the order of operations is crucial. In extreme situation an event can be delivered faster than value is set, and potentially other thread can access old value with *Get* while value should already be new one. In situation where some synchronization mechanism is used such scenario is not applicable. See a chapter about synchronization mechanisms in DPL.

GENERIC EVENTS

To easily declare an event, there are provided several useful macros. Developer is not obliged to use them. Instead of using them, custom structures can be used, but usually they are mechanical and impractical (although they may have custom argument names).

Example:

```
#include <dpl/generic_event.h>

DECLARE_GENERIC_EVENT_0(EmptyEvent)
DECLARE_GENERIC_EVENT_0(OtherEmptyEvent)
DECLARE_GENERIC_EVENT_1(IntegerEvent, int)
DECLARE_GENERIC_EVENT_5(BigEvent,
                        std::string, int,
                        MyStruct, const char *, double)
```

In example, we declare four different events. There are two different empty events, one event which contains one integer and a “big” event which contains several different fields.

© Samsung Electronics Co., Confidential

To declare an event with *k* fields we use macro DECLARE_GENERIC_EVENT_*k*. By default DPL support events with at most 8 parameters. In practice it is sufficient, because a large amount of parameter would be messy. In practice, if we have more than three or four parameters, usually, there is already a structure that contains them all, and this structure is used as a field in event.

Notice:

```
#include <dpl/generic_event.h>

DECLARE_GENERIC_EVENT_0(EventOne)   !=   DECLARE_GENERIC_EVENT_0(EventTwo)
```

That means, even if declarations are the same, the events are different types and can be used simultaneously.

Notice:

To avoid extra compilation warnings, do not put semicolon after generic event declaration:

```
#include <dpl/generic_event.h>

DECLARE_GENERIC_EVENT_0(SampleEvent);
DECLARE_GENERIC_EVENT_0(SampleEvent)
```

To access event fields, there are provided methods:

GetArg0(), *GetArg1*(), *GetArg2*(), … *GetArg*{K-1}, K is a number of fields that are defined in generic event.

It is possible to set event field value with *SetArgN* methods, but they are rarely used.

MODELS

With basic event support template with are able to construct one of the most important concepts in MVC – a model. One of many possible interpretations is that a model is that a model is a data container that supports listening for its contents changes. Of course, with DPL's basic templates it is possible to build other interpretations of MVC model.

Example:

```
#include <dpl/generic_event.h>
#include <dpl/event_support.h>

#include <string>

DECLARE_GENERIC_EVENT(NameChangedEvent, std::string)

class User : public DPL::EventSupport<NameChangedEvent>
{
private:
    std::string m_name;

public:
    void GetName() const
    {
```


© Samsung Electronics Co., Confidential

```
        return m_name;
    }

    void SetName(const std::string &name)
    {
        if (m_name == name)
            return;

        m_name = name;

        DPL::EventSupport<NameChangedEvent>::
            EmitEvent(NameChangedEvent(m_name, this));
    }
};
```

A model from example represents a User. It contains only one field – a name. With event support one can listen for user's name changes. After user name change an event is emitted: NameChangedEvent.

Models are usually used in threaded environment. To see how to synchronize reading and writing data to a model, go to chapter about DPL's support for synchronization. ReadWriteMutex is a recommended method.

CONTROLLERS

On top of basic event support, DPL provides support for controllers. In DPL controller is an object that is capable of receiving events from any thread and executing them in selected thread. Target thread can be selected at runtime and it can be both custom thread and main (EFL or GTK) thread.

Basic example use of controller will be discussed on an example.

Example:

```
#include <dpl/generic_event.h>
#include <dpl/controller.h>
#include <dpl/application.h>

DECLARE_GENERIC_EVENT_0(FirstEvent)
DECLARE_GENERIC_EVENT_0(SecondEvent)

class ControllerInThread : public DPL::Controller2<FirstEvent, SecondEvent>
{
protected:
    virtual void OnEventReceived(const FirstEvent &event) { }
    virtual void OnEventReceived(const SecondEvent &event) { }
};


DECLARE_GENERIC_EVENT_0(QuitEvent)

class MyApplication
    : public DPL::Application,
      private DPL::Controller1<QuitEvent>
{
    DPL::Thread m_thread;
    ControllerInThread m_controllerInThread;

    virtual void OnEventReceived(const QuitEvent &event)
    {
```


© Samsung Electronics Co., **Confidential**

```
        Quit();
    }

public:
    MyApplication(int argc, char **argv) : Application(argc, argv)
    {
        Touch();
        m_controllerInThread.Touch();

        m_thread.Run();
        m_controllerInThread.SwitchToThread(&m_thread);
        m_controllerInThread.DPL::ControllerEventHandler<SecondEvent>::
            PostTimedEvent(SecondEvent());
        m_controllerInThread.DPL::ControllerEventHandler<FirstEvent>::
            PostEvent(FirstEvent());
        DPL::ControllerEventHandler<QuitEvent>::PostEvent(QuitEvent());
    }
    virtual ~MyApplication()
    {
        m_controllerInThread.SwitchToThread(NULL);
        m_thread.Quit();
    }
};


int main(int argc, char *argv[])
{
    MyApplication app(argc, argv);
    return app.Exec();
}
```

Creating and using DPL object which are subject of context inheritance will always be given context from controller. See chapter about context inheritance for additional information.

CONTEXT INHERITANCE

Some of DPL objects are subject of context inheritance (all sockets, semaphores, named pipes and other). For proper execution they need a working context (thread) to register for events or other low-level signals. Upon receiving such event they start processing of it in selected thread. If an object inherits calling context it means that is saves current thread identifier and uses it in later execution as an environment to process asynchronous calls.

Note:

Carefully select which worker threads and which objects should be held together as a result of context inheritance. Also remember if such object cannot be created in target thread, single listeners can be manually switched to target thread (see advanced topics).

ADVANCED TOPICS

Advanced topics cover mainly implementation details. Standard library usages normally do not need developer to get into advanced topics.

MANUAL SWITCH OF THREAD INHERITED BY EVENT SUPPORT

© Samsung Electronics Co., **Confidential**

In some scenarios it is difficult to create an object with context inheritance or to add listener directly in target thread. It is possible then to create object or add listener in any thread and then switch inherited context to selected one. It can be easily done with:

Example:

```
#include <dpl/thread.h>
#include <dpl/event_support.h>

DPL::Thread otherThread;

someClass->DPL::EventSupport<SomeEvent>::AddListener(this);

someClass->DPL::EventSupport<SomeEvent>::
    SwitchAllListenersToThread(&otherThread);
```

If any events are already waiting to be delivered, they will be reposted to target thread (so called ping-pong scenario).

GENERIC EVENT CALL

Generic event call is a template that creates functor that calls listener 'receive' method, passing represented event. Do not use generic event calls directly, as they are internal and always subject to change.

INTERNAL REPRESENTATION OF EVENT

Internally, an event is represented as an abstract call. Its implementations are derived via template-based genetic event call.

THREAD SUPPORT

DPL gives a wide support for event based threads.

THREAD

To provide multithreaded MVC environment, thread with event loop is needed. In DPL there is an implementation of thread with event support. DPL Thread object is a wrapper for POSIX thread, but also gives a possibility to send events and process them in thread.

A standard usage of DPL thread normally ends up with granting controllers a dedicated thread environment.

Example:

```
#include <dpl/thread.h>

DPL::Thread *g_threadDedicated = NULL;

void GrantControllers()
{
    g_threadDedicated = new DPL::Thread();

    g_threadDedicated->Run();
```


© Samsung Electronics Co., Confidential

```
    SomeControllerSingleton::Instance().SwitchToThread(g_threadDedicated);
}

void UnGrantCOntrollers()
{
    SomeControllerSingleton::Instance().SwitchToThread(NULL);

    g_threadDedicated->Quit();

    delete g_threadDedicated;
    g_threadDedicated = NULL;
}
```

Remember that there should be no controllers with attached thread when *Thread* object is destroyed. Always detach threads from controllers upon exit.

ADVANCED TOPICS

WAITABLE HANDLE WATCH SUPPORT

Waitable handle watch support gives object an ability to provide mechanism to watch waitable handles in selected execution context.

WAITABLE INPUT OUTPUT EXECUTION CONTEXT SUPPORT

Waitable input/output execution context support gives object an ability to automatically submit data to abstract waitable output and automatically read data from abstract input.

ABSTRACT IO

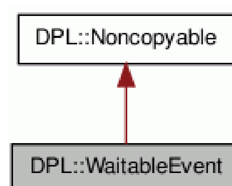
DPL gives a great abstraction of IO layer. In practice it gives endless possibilities of inter-object configurations and connections. Whole abstraction layer for IO is build on top of event based architecture and supports context inheritance.

WAITABLE EVENT

Many of synchronization mechanisms are performed with use of WaitableEvent object. Basically, waitable event object is a primitive, which has a property that it can be waited for it to be signaled. During wait, current thread is blocked. It is similar to a semaphore, but more lightweight option.

Waitable event can be in one of two possible states: signaled and not signaled. After creation, by default it is not signaled.

Inheritance diagram:



Waitable event usage:

© Samsung Electronics Co., Confidential

Method or call	Result
WaitableEvent::Reset	Resets a signaled state of waitable event. All waiting threads are blocked.
WaitableEvent::Signal	Signals a waitable event. All waiting threads are released
WaitForSingleHandle / WaitForMultipleHandles	Blocking wait for single or multiple waitable handles. Returns a list of signaled handle indexes.

Low level waitable handle is implementation dependant.

Typical usage, execution of asynchronous operation and waiting for a result is as follows:

Example:

```
DPL::WaitableEvent doneEvent;
ResultStruct result;

// Post event to some controller
CONTROLLER_POST_EVENT(SomeController,
                      DoSomething(&doneEvent, &result));

DPL::WaitForSingleHandle(doneEvent.GetHandle());
```

To wait for multiple events at the same time, use *WaitForMultipleHandles* function.

Example:

```
DPL::WaitableEvent eventOne;
DPL::WaitableEvent eventTwo;
DPL::TcpSocket sock;

WaitableHandleList handles;
handles.push_back(eventOne.GetHandle());
handles.push_back(eventTwo.GetHandle());
handles.push_back(sock.GetReadHandle());

WaitableHandleIndexList indexes =
    DPL::WaitForSingleHandle(handles);
```

It is important to properly handle result of multiple handle waiting routine.

Remember:

More than one event may be signaled. Remember to handle all events that indexes were returned. If one fails to do so, a ***starvation*** may occur.

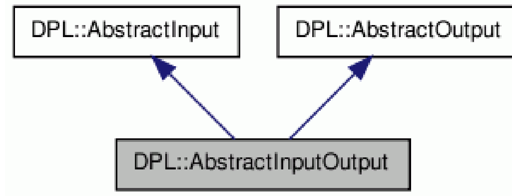
To wait simultaneously for both read and write events from socket, a more advanced function must be used: *WaitFormMultipleHandles* with list of type *WaitableHandleListEx*. For each element, an additional flag is given whether we wait for readability of writability.

ABSTRACT INPUT/OUTPUT

© Samsung Electronics Co., Confidential

On top of abstraction for IO, there is an abstract input/output object. It is a composition of abstract input object and abstract output object. For dynamic data operation, in all buffer operations, BinaryQueue object is used, which is a implementation of fast buffer operations.

Inheritance diagram:



Abstract input and output interfaces:

Interface	Method	Description
Abstract input	BinaryQueueAutoPtr Read(size_t size)	Read a maximum *size* number of bytes from abstract input. Returns a *BinaryQueue* or NULL. Empty queue is returned when connection was gracefully closed and NULL is returned when no data is currently available. Method, depending on implementation, can additionally throw *ReadFailed* exceptions and other.
Abstract output	size_t Write(const BinaryQueue &buffer, size_t bufferSize)	Write data from binary queue. Write at most *bufferSize* bytes. Return number of bytes actually wrote or zero when connection is blocked. Upon error, an exception can be thrown depending on implementation.

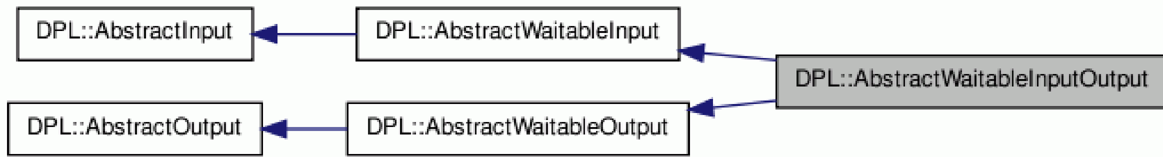
If abstract input returns null binary queue it means that there is no data waiting. There is no way to wait for incoming data. Only simple timeout checks are possible. To make waiting possible, use abstract waitable input interfaces. The same applies to abstract output and waiting for writability.

ABSTRACT WAITABLE INPUT/OUTPUT

Abstract waitable input output is an abstract interface that besides being an abstract input/output it adds functionality of waiting for data to read or send.

Inheritance diagram:

© Samsung Electronics Co., Confidential

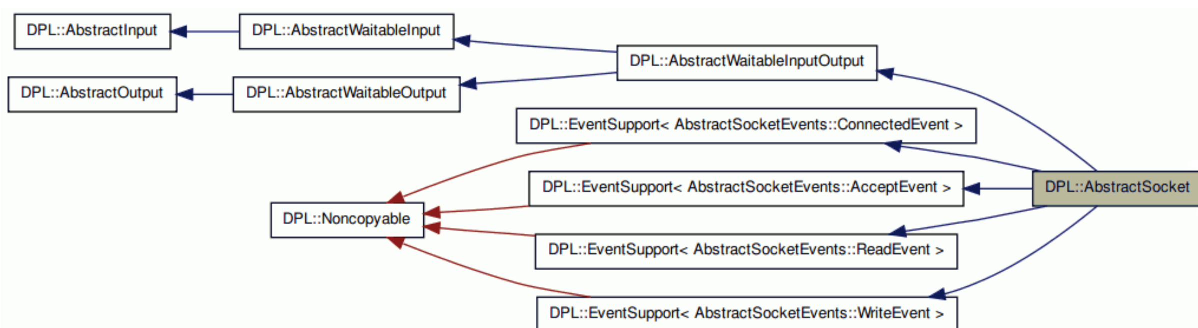


Both interfaces additionally define waitable handle which can be used for waiting. To retrieve a waitable handle call *WaitableWriteHandle() / WaitableReadHandle().*

ABSTRACT SOCKET

Abstract socket is an abstract socket for all socket based interfaces. Abstract socket support four base events: connected event, accept event, read event and write event.

Inheritance diagram:



DPL sockets are asynchronous. They support a set of basic event:

Abstract socket event	Meaning
AbstractSocketEvents::ConnectedEvent	Socket connected to remote host
AbstractSocketEvents::AcceptEvent	A remote connection is waiting to be accepted. In accept handler one should call *Accept()* method to accept connection. Unwanted connections may be immediately closed by deleting new connection object.
AbstractSocketEvents::ReadEvent	An incoming data is waiting to be read. In read handler one should call *Read()* method to read data.
AbstractSocketEvents::WriteEvent	Socket is ready for writing more data. This event will be emitted only once, after socket was blocked and recovered from.

A basic set of methods are defined:

© Samsung Electronics Co., Confidential

Abstract socket method	Description
void Connect(const Address &address)	Begin asynchronous connecting to remote host. After connection is established a corresponding event is emitted. Current context is inherited.
void Open()	Open a socket for usage. Must be called first.
void Close()	Close a socket and all connections that it represents.
void Bind(const Address &address)	Bind an address to a socket
void Listen(int backlog)	Begin listening on a socket. An address must be bound before this method is called. Backlog is a suggested size of waiting connection queue.
AbstractSocket *Accept()	Accept a waiting socket. Usually called from accept event handler. If no socket can be accepted it returns NULL.
Address GetLocalAddress()	Get an address of socket local endpoint
Address GetRemoteAddress()	Get an address of socket remote endpoint

In all socket related mechanisms, an Address object is used to represent socket addresses.

ADDRESS

An address can represent a socket local or remote endpoint. It is a pair of string description and port number. In UNIX sockets port number is ignores. In TCP sockets, address string is translated with a DNS request.

Example:

```
#include <dpl/address.h>

DPL::Address address("www.somewhere.com", 8080);

unsigned short port = address.GetPort();
std::string name = address.GetAddress();
std::string fullName = address.ToString();
```

A convenience method *ToString()* returns an address in form *address:port*.

GENERIC SOCKET

© Samsung Electronics Co., Confidential

All socket types are implemented via template based generic socket. To implement custom socket type, it is convenient to use basic implementation provided by generic socket.

UNIX/TCP SOCKET

UNIX and TCP sockets are specific implementations for TCP and UNIX protocols. Both are derived from generic socket implementation.

Following example is a simple HTTP client displaying default page to *stdout*.

Example:

```
#include <dpl/tcp_socket.h>
#include <dpl/abstract_socket.h>
#include <dpl/application.h>
#include <dpl/generic_event.h>
#include <dpl/binary_queue.h>
#include <dpl/scoped_array.h>
#include <dpl/log.h>
#include <string>
#include <cassert>

class MyApplication
    : public DPL::Application,
      private DPL::EventListener<
                        DPL::AbstractSocketEvents::ConnectedEvent>,
      private DPL::EventListener<
                        DPL::AbstractSocketEvents::ReadEvent>
{
private:
    DPL::TcpSocket m_socket;

    virtual void OnEventReceived(
        const DPL::AbstractSocketEvents::ConnectedEvent &event)
    {
        (void)event;
        LogInfo("Connected!");

        // Send request
        DPL::BinaryQueue data;
        const char *query =
            "GET /wiki/Main_Page HTTP/1.1\nHost: en.wikipedia.org\n\n";
        data.AppendCopy(query, strlen(query) + 1);
        m_socket.Write(data, data.Size());
    }

    virtual void OnEventReceived(
        const DPL::AbstractSocketEvents::ReadEvent &event)
    {
        (void)event;
        LogInfo("Read!");

        DPL::BinaryQueueAutoPtr data = m_socket.Read(100);
```


© Samsung Electronics Co., Confidential

```
        assert(data.get() != NULL);

        if (data->Empty())
        {
            LogInfo("Connection closed!");
            m_socket.Close();

            // Done
            Quit();
            return;
        }

        // Show data
        DPL::ScopedArray<char> text(new char[data->Size()]);
        data->Flatten(text.Get(), data->Size());

        LogPedantic("READ: \n" <<
                    std::string(text.Get(), text.Get() + data->Size()) <<
                    "\n");
    }

public:
    MyApplication(int argc, char **argv)
        : Application(argc, argv, "tcpsock")
    {
        LogInfo("CTOR!");

        // Add listeners
        m_socket.DPL::EventSupport<
            DPL::AbstractSocketEvents::ConnectedEvent>::AddListener(this);
        m_socket.DPL::EventSupport<
            DPL::AbstractSocketEvents::ReadEvent>::AddListener(this);

        // Connect
        m_socket.Open();
        LogInfo("Connecting...");
        m_socket.Connect(DPL::Address("en.wikipedia.org", 80));
    }

    virtual ~MyApplication()
    {
        LogInfo("DTOR!");

        // Remove listeners
        m_socket.DPL::EventSupport<
            DPL::AbstractSocketEvents::ConnectedEvent>::
                RemoveListener(this);
        m_socket.DPL::EventSupport<
            DPL::AbstractSocketEvents::ReadEvent>::
                RemoveListener(this);
    }
};
```


© Samsung Electronics Co., Confidential

```
int main(int argc, char *argv[])
{
    MyApplication app(argc, argv);
    return app.Exec();
}
```

To download a page, first a socket is opened with *Open()*. After that, connection phase is initialized with *Connect()* method. If connected event handler was added, it is possible to check when connection is established. After connection has been established, a buffer with http request is immediately sent to socket. This can be easily done with constructing proper *BinaryQueue* object and passing it to socket Write method.
After page request, all incoming data is received through read event handler. Note that, if we ignore once read data event, even if data is available, and no more data arrives, a read event will not be emitted again. In other words, read event is only generated once for each portion of incoming bytes. If we receive empty binary queue (not null) it means that connection was gracefully closed. Example application exits then.

SYNCHRONIZATION

Several synchronization mechanisms are provided in DPL to get all benefits from threaded environment. The simplest is mutex, more specialized are recursive mutex and spinlock.

MUTEX

Basically DPL Mutex is a wrapper for pthread mutex, but it is managed. Most important feature is scoped locking mechanism. It means that it is only possible to lock a mutex for selected scope, by using scoped lock object. It is impossible to call Lock/Unlock routines directly. This is an example of RAII object.

Example:

```
#include <dpl/mutex.h>

DPL::Mutex g_mutex;

void Foo()
{
    DPL::Mutex::ScopedLock lock(&g_mutex);
    Bar();
}
```

In example above, function *Bar* is synchronized with *g_mutex*. A mutex is held during all function body.

RECURSIVE MUTEX

Recursive mutex concept is similar to standard mutex. It has one more feature – it can be locked multiple times in the same thread and this will not cause deadlock. Internally recursive mutex holds a number of recursive locks. As standard mutex has scoped lock object, similar scoped lock object is in recursive mutex.

Example:

```
#include <dpl/recursive_mutex.h>

DPL::RecursiveMutex g_mutex;

```


© Samsung Electronics Co., Confidential

```
void Nested()
{
    DPL::RecursiveMutex::ScopedLock lock(&g_mutex);
    Bar();
}

void Foo()
{
    DPL::RecursiveMutex::ScopedLock lock(&g_mutex);
    Nested();
}
```

Note, that recursive mutex is more powerful than simple mutex. Recursive mutex has more complicated implementation, and thus it should be used only where applicable.

SPIN LOCK

Spin lock is a very light synchronization mechanism. Locking/Unlocking is done with single assembler instructions. The main disadvantage is that if scoped lock cannot be taken it is blocking current thread actively. In practice it means that whole process is using maximum CPU resources. It is useful in situations, where probability of collision is very low and scoped routine under lock is very short. Interface of spin lock is exactly the same as simple Mutex.

Example:

```
#include <dpl/spin_lock.h>

DPL::SpinLock g_mutex;


void Foo()
{
    DPL::SpinLock::ScopedLock lock(&g_mutex);
    value = currentValue + deltaValue;
}
```

READ WRITE MUTEX

It is inefficient to lock whole model by simple mutex to read some data from it. After deeper analysis of locking mechanism one can be deduce that only exclusive write lock is needed while many simultaneous read can occur at the same time. To make profit from such observation, special locking mechanism is provided – ReadWriteMutex. It is a mutex that have two different scoped locks. One is for exclusive write, and one for simultaneous reads.

Example:

```
#include <dpl/read_write_mutex.h>

class SimpleModel
{
    mutable DPL::ReadWriteMutex m_mutex;
```


© Samsung Electronics Co., Confidential

```
    std::string m_value;

public:
    std::string GetValue() const
    {
        DPL::ReadWriteMutex::ScopedReadLock lock(&m_mutex);
        return m_value;
    }

    void SetValue(const std::string &value)
    {
        DPL::ReadWriteMutex::ScopedWriteLock lock(&m_mutex);
        m_value = value;
    }
};
```

REMOTE PROCEDURE CALLS

DPL has a wide support for RPC calls. Similarly to abstract IO, RPC subsystem has got a wide abstraction. It is built on top of abstract IO, and therefore can use various transports, such as: generic sockets (TCP, UNIX) or pipes.

ABSTRACT RPC CONNECTION

Abstract RPC connection represents a connected pair of clients which can communicate with RPC calls. Abstract RPC connection is delivered through connected event from RPC connector.

RPC connection allows invoking RPC calls. These are intended to behave similarly to normal C++ functions.

RPC CALL

All data posted through RPC connection is encoded as a RPC call. When posting an event, *RPCFunction* object is created and arguments are pushed with template based *AppendArg* method.

Example:

```
#include <dpl/rpc_function.h>

enum SampleEnum { SampleEnum_Value };

DPL::RPCFunction func;
func.AppendArg(SampleEnum_Value);
func.AppendArg(12345);
func.AppendArg(std::string("This is a test string"));

connection->AsyncCall(func);
```


© Samsung Electronics Co., Confidential

To receive RPC asynchronous call, RPC function must be parsed. This can be easily done with *ConsumeArg* which opposite to *AppendArg* method. Consecutive calls to ConsumeArg will return saved parameters beginning from first one pushed (FILO).

Example:

```
#include <dpl/rpc_function.h>

enum SampleEnum { SampleEnum_Value };

DPL::RPCFunction func;

SampleEnum status;
func.ConsumeArg(status);
int value;
func.ConsumeArg(value);
std::string name;
func.ConsumeArg(name);
```

There is no way to check at runtime what types of arguments are placed in RPC call. Nevertheless pushed arguments sizes are also saved and simple *sizeof* checks are always performed at runtime.

Remember:

Carefully add any pointers to buffer because of template nature of this AppendArg method. All arguments are always copied by value, thus no deep copy on pointers is performed.

GENERIC SOCKET RPC CONNECTION

Generic socket RPC connection implements RPC connection with use of abstract socket.

UNIX/TCP SOCKET RPC CONNECTION

UNIX and TCP socket implementations for RPC connection are template derived implementations for specific abstract sockets. When using corresponding RPC client/server implementation for Unix or Tcp sockets, one can cast abstract RPC connection up to Unit or TCP connection. In practice it is usually not needed, because level of abstraction is decreased.

ABSTRACT RPC CONNECTOR

To establish a RPC connection one must use a connector. A connector can be a server, client or some special type. Connector inherits execution context to establish connection. Abstract RPC connector support one event: *ConnectionEstablisedEvent.* The event must be handled. It contains Abstract RPC connection ID, which is an ID given at the moment of executing opening phase for connector.

Note:

All connectors support simultaneous opening of many connections. All simultaneous connections are given different ID, by which they can be distinguished in connection established event handler.

© Samsung Electronics Co., **Confidential**

Along with connection ID, if connection succeeds, abstract RPC connection is also passed in event handler. The pointer with connection must be managed and deleted later. It is usually convenient to use some smart pointer, such as *ScopedPtr* or *SharedPtr*. After receiving new connection, it is usually desirable to immediately connect all abstract connection event handlers.

Example:

```
#include <dpl/abstract_rpc_connection.h>

DPL::ScopedPtr<DPL::AbstractRPCConnection> m_rpcConnection;

void Test::OnEventReceived(const DPL::AbstractRPCConnectorEvents::
                           ConnectionEstablishedEvent &event)
{
    // Save connection pointer
    m_rpcConnection.Reset(event.GetArg1());

    // Attach listener to new connection
    m_rpcConnection->DPL::EventSupport<
        DPL::AbstractRPCConnectionEvents::AsyncCallEvent>::
            AddListener(this);
    m_rpcConnection->DPL::EventSupport<
        DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent>::
            AddListener(this);
    m_rpcConnection->DPL::EventSupport<
        DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent>::
            AddListener(this);
}
```

GENERIC SOCKET RPC CLIENT/SERVER

UNIX/TCP SOCKET RPC CLIENT/SERVER

RPC PING-PONG EXAMPLE

A simple ping-pong RPC example is provided for better understanding of RPC usage. RPC is based on UNIX sockets, but can be easily adopted into TCP with just a few changes.

Example:

```
#include <dpl/unix_socket_rpc_client.h>
#include <dpl/unix_socket_rpc_server.h>
#include <dpl/unix_socket_rpc_connection.h>
#include <dpl/scoped_ptr.h>
#include <dpl/application.h>
#include <dpl/controller.h>
#include <dpl/thread.h>
```


© Samsung Electronics Co., Confidential

```
#include <dpl/log.h>
#include <string>

static const char *RPC_NAME = "/tmp/unix_socket_rpc";

class MyThread
    : public DPL::Thread,
      private DPL::EventListener<
          DPL::AbstractRPCConnectionEvents::AsyncCallEvent>,
      private DPL::EventListener<
          DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent>,
      private DPL::EventListener<
          DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent>,
      private DPL::EventListener<
          DPL::AbstractRPCConnectorEvents::ConnectionEstablishedEvent>
{
private:
    DPL::UnixSocketRPCClient m_rpcClient;
    DPL::ScopedPtr<DPL::AbstractRPCConnection> m_rpcConnection;

    virtual void OnEventReceived(
        const DPL::AbstractRPCConnectionEvents::AsyncCallEvent &event)
    {
        (void)event;

        LogInfo("CLIENT: AsyncCallEvent received");

        int value;
        event.GetArg0().ConsumeArg(value);
        LogInfo("CLIENT: Result from server: " << value);
    }

    virtual void OnEventReceived(
      const DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent &event)
    {
        (void)event;
        LogInfo("CLIENT: ConnectionClosedEvent received");
    }

    virtual void OnEventReceived(
      const DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent &event)
    {
        (void)event;
        LogInfo("CLIENT: ConnectionBrokenEvent received");
    }

    virtual void OnEventReceived(
        const DPL::AbstractRPCConnectorEvents::
            ConnectionEstablishedEvent &event)
    {
        // Save connection pointer
        LogInfo("CLIENT: Acquiring new connection");
        m_rpcConnection.Reset(event.GetArg1());
```


© Samsung Electronics Co., Confidential

```
        // Attach listener to new connection
        LogInfo("CLIENT: Attaching connection event listeners");
        m_rpcConnection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::AsyncCallEvent>::
                AddListener(this);
        m_rpcConnection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent>::
                AddListener(this);
        m_rpcConnection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent>::
                AddListener(this);

        LogInfo("CLIENT: Connection established");

        // Emit RPC function call
        DPL::RPCFunction proc;
        proc.AppendArg((int)1111);
        LogInfo("CLIENT: Calling RPC function");
        m_rpcConnection->AsyncCall(proc);
    }

public:
    virtual ~MyThread()
    {
        // Always quit thread
       Quit();
    }

    virtual int ThreadEntry()
    {
        // Attach RPC listeners
        LogInfo("CLIENT: Attaching connection established event");
        m_rpcClient.DPL::EventSupport<
            DPL::AbstractRPCConnectorEvents::ConnectionEstablishedEvent>::
                AddListener(this);

        // Open connection to server
        LogInfo("CLIENT: Opening connection to RPC");
        m_rpcClient.Open(RPC_NAME);

        // Start message loop
        LogInfo("CLIENT: Starting thread event loop");
        int ret = Exec();

        // Detach RPC listeners
        if (m_rpcConnection.Get())
        {
            LogInfo("CLIENT: Detaching RPC connection events");
            m_rpcConnection->DPL::EventSupport<
                DPL::AbstractRPCConnectionEvents::AsyncCallEvent>::
                    RemoveListener(this);
            m_rpcConnection->DPL::EventSupport<
```


© Samsung Electronics Co., Confidential

```
                DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent>::
                    RemoveListener(this);
            m_rpcConnection->DPL::EventSupport<
                DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent>::
                    RemoveListener(this);

            LogInfo("CLIENT: Resetting connection");
            m_rpcConnection.Reset();
        }

        // Detach RPC client listener
        LogInfo("CLIENT: Detaching connection established event");
        m_rpcClient.DPL::EventSupport<
            DPL::AbstractRPCConnectorEvents::ConnectionEstablishedEvent>::
                RemoveListener(this);

        // Close RPC
        LogInfo("CLIENT: Closing RPC client");
        m_rpcClient.CloseAll();

        // Done
        return ret;
    }
};

DECLARE_GENERIC_EVENT_0(QuitEvent)
DECLARE_GENERIC_EVENT_0(CloseThreadEvent)

class MyApplication
    : public DPL::Application,
      private DPL::Controller2<QuitEvent,
                               CloseThreadEvent>,
      private DPL::EventListener<
          DPL::AbstractRPCConnectionEvents::AsyncCallEvent>,
      private DPL::EventListener<
          DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent>,
      private DPL::EventListener<
          DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent>,
      private DPL::EventListener<
          DPL::AbstractRPCConnectorEvents::ConnectionEstablishedEvent>
{
private:
    DPL::UnixSocketRPCServer m_rpcServer;
    DPL::ScopedPtr<DPL::AbstractRPCConnection> m_rpcConnection;

    MyThread m_thread;

    // Quit application event occurred
    virtual void OnEventReceived(const QuitEvent &event)
    {
        (void)event;
        Quit();
    }
```


© Samsung Electronics Co., Confidential

```
virtual void OnEventReceived(const CloseThreadEvent &event)
{
    (void)event;
    m_thread.Quit();
}

virtual void OnEventReceived(
    const DPL::AbstractRPCConnectionEvents::AsyncCallEvent &event)
{
    (void)event;

    LogInfo("SERVER: AsyncCallEvent received");

    int value;
    event.GetArg0().ConsumeArg(value);
    LogInfo("SERVER: Result from client: " << value);
}

virtual void OnEventReceived(
  const DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent &event)
{
    (void)event;

    LogInfo("SERVER: ConnectionClosedEvent received");

    // Close RPC now
    LogInfo("SERVER: Closing RPC connection on event...");

    // Detach RPC connection listeners
    if (m_rpcConnection.Get())
    {
        LogInfo("SERVER: Detaching connection events");
        m_rpcConnection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::AsyncCallEvent>::
                RemoveListener(this);
        m_rpcConnection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent>::
                RemoveListener(this);
        m_rpcConnection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent>::
                RemoveListener(this);
        m_rpcConnection.Reset();
    }
    LogInfo("SERVER: RPC connection closed");

    LogInfo("SERVER: Closing RPC on event...");
    m_rpcServer.CloseAll();
    LogInfo("SERVER: RPC closed");
}

virtual void OnEventReceived(
  const DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent &event)
```


© Samsung Electronics Co., Confidential

```
    {
        (void)event;
        LogInfo("SERVER: ConnectionBrokenEvent received");
    }

    virtual void OnEventReceived(
        const DPL::AbstractRPCConnectorEvents::
            ConnectionEstablishedEvent &event)
    {
        // Save connection pointer
        LogInfo("SERVER: Acquiring RPC connection");
        m_rpcConnection.Reset(event.GetArg1());

        // Attach event listeners
        LogInfo("SERVER: Attaching connection listeners");
        m_rpcConnection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::AsyncCallEvent>::
                AddListener(this);
        m_rpcConnection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent>::
                AddListener(this);
        m_rpcConnection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent>::
                AddListener(this);

        LogInfo("SERVER: Connection established");

        // Emit RPC function call
        DPL::RPCFunction proc;
        proc.AppendArg((int)2222);
        LogInfo("SERVER: Calling RPC function");
        m_rpcConnection->AsyncCall(proc);
    }

public:
    MyApplication(int argc, char **argv)
        : Application(argc, argv, "rpc")
    {
        // Attach RPC server listeners
        LogInfo("SERVER: Attaching connection established event");
        m_rpcServer.DPL::EventSupport<
            DPL::AbstractRPCConnectorEvents::ConnectionEstablishedEvent>::
                AddListener(this);

        // Self touch
        LogInfo("SERVER: Touching controller");
        Touch();

        // Open RPC server
        LogInfo("SERVER: Opening server RPC");
        m_rpcServer.Open(RPC_NAME);

        // Run RPC client in thread
```


© Samsung Electronics Co., Confidential

```
        LogInfo("SERVER: Starting RPC client thread");
        m_thread.Run();

        // Quit application automatically in few seconds
        LogInfo("SERVER: Sending control timed events");
        DPL::ControllerEventHandler<CloseThreadEvent>::
            PostTimedEvent(CloseThreadEvent(), 2);
        DPL::ControllerEventHandler<QuitEvent>::
            PostTimedEvent(QuitEvent(), 3);
    }

    virtual ~MyApplication()
    {
        // Quit thread
        LogInfo("SERVER: Quitting thread");
        m_thread.Quit();

        // Close RPC server
        LogInfo("SERVER: Closing RPC server");
        m_rpcServer.CloseAll();

        // Detach RPC server listener
        m_rpcServer.DPL::EventSupport<
            DPL::AbstractRPCConnectorEvents::ConnectionEstablishedEvent>::
                RemoveListener(this);
    }
};

int main(int argc, char *argv[])
{
    LogInfo("Starting");
    MyApplication app(argc, argv);
    return app.Exec();
}
```

In example code comments are provided to each source fragment. Reader is encouraged to go through code for better understanding of RPC mechanism.

LOGGING SYSTEM

DPL unifies various logging systems. By default DLOG subsystem is used. This can be changed by attaching custom log provider. Logging system uses stream based message construction, thus making logging very easy and convenient.

Example:

```
#include <dpl/log.h>

int number;
std::string msg1;
const char *msg2;

LogInfo("Testing: " << msg1 << " and " << number << " other " << msg2);
```


© Samsung Electronics Co., **Confidential**

```
LogWarning("Computations may be inaccurate");
LogDebug("Current height is: " << height);
LogError("Division by zero occurred!");
```

In DPL there are implemented default log providers. The first one – DLOG log provider uses DLOG subsystem to log messages, and second one – old style that uses plain *printf* based logging system.

There are four log categories:

- *Info* – used to inform about general routines that are taking place
- *Debug* – detailed information about parameters of routines taking place and other optional information
- *Warning* – messages that inform about a recoverable problems
- *Error* – messages about failures

To attach new log provider AbstractLogProvider must be implemented and attached with *AddProvider*() from LogSystem. See example above.

Example:

```
#include <dpl/log.h>

class MyLogProvider
{
public:
    virtual void Debug(const char *message, const char *fileName,
                       int line, const char *function)
    {
        std::cout << "Got debug message: " << message << std::endl;
    }

    virtual void Info(const char *message, const char *fileName,
                      int line, const char *function) {}
    virtual void Warning(const char *message, const char *fileName,
                         int line, const char *function) {}
    virtual void Error(const char *message, const char *fileName,
                       int line, const char *function) {}
    virtual void Pedantic(const char *message, const char *fileName,
                          int line, const char *function) {}
};

LogSystemSingleton::Instance().AddProvider(new MyLogProvider());
```

Note that upon adding new log provider responsibility for its deletion is transferred to *LogSystem* singleton. Pedantic log category is used internally by DPL and is <u>not intended</u> to use outside DPL. Pedantic logs are extremely verbose and are used for debugging DPL or hard debugging application using DPL. To see pedantic log output provide custom log provider with implementation of *Pedantic* method or use old style log provider and set proper environment variables.

DLOG PROVIDER

DLOG log provider directly passes all log messages to DLOG library. Note that DLOG library log categories slightly differ from DPL log categories. They are translated according to DLOG translation guide.

© Samsung Electronics Co., Confidential

Generally, DLOG log provider performance is lower than old style log provider. The reason of that is that DLOG log provider uses DLOG and which uses interprocess communication via unix sockets. So every message must first be transferred through unix socket before returning from logging routine. When process is overloaded this can decrease logging performance.

OLD STYLE PROVIDER

Old style log provider is fast *printf* based log provider. By default it is disabled. Following environment macros can be defined to enable old style log provider and along with that disable default DLOG log provider.

Environment variable	Action	Settings
DPL_USE_OLD_STYLE_LOGS	Enable old style log provider instead of DLOG log provider. This does not enable pedantic logs.	Default: *undefined* Enable: *set to 1*
DPL_USE_OLD_STYLE_PEDANTIC_LOGS	Enable pedantic logs display. Note that DPL_USE_OLD_STYLE_LOGS must be also enabled to see logs.	Default: *undefined* Enable: *set to 1*
DPL_USE_OLD_STYLE_LOGS_MASK	Set visible log categories. Does not include pedantic logs.	Default: undefined Enable: *set to ABCD where A, B, C, D are 0 or 1 and they correspond to Info, Debug, Warning, Error categories. Example: 1010 is: Info=on, Debug=off, Warning=on, Error = off*

To disable a previously set feature undefined its environment variable or set it to zero.

Note that it is also possible to have both DLOG and old style log provider enabled at the same time. By default DLOG log provider is added. You should not define any old style control variables, but attach in application code new old style log provider as follows:

Example:

```
#include <dpl/log.h>
#include <dpl/old_style_log_provider.h>

int main(int argc, char *argv[])
{
    DPL::LogSystemSingleton::Instance().AddProvider(
        new DPL::OldStyleLogProvider());
    […]
}
```

There can be attached any number of log providers to log system. No particular order should be assumed according to calls to them during logging a message.

© Samsung Electronics Co., Confidential

EXCEPTION SYSTEM

In DPL we can use rich exception system that is an extension to standard C++ exception system. Not only exception type is known after exception catch, but also origin of exception and full exception stack. This is very useful after accidental exception in application. DPL also provider unhandled exception handlers to ensure that exceptions are not leaking out of application causing abnormal exits.

DPL has its own exception hierarchy. It is parallel to object hierarchy. By convention, all object exceptions are defined as class elements of namespace Exception in object. See example below.

Example:

```
namespace DPL
{
class SqlConnection
{
public:
    class Exception
    {
    public:
        DECLARE_EXCEPTION_TYPE(DPL::Exception, Base)
        DECLARE_EXCEPTION_TYPE(Base, SyntaxError)
        DECLARE_EXCEPTION_TYPE(Base, ConnectionBroken)
        DECLARE_EXCEPTION_TYPE(Base, InternalError)
        DECLARE_EXCEPTION_TYPE(Base, InvalidColumn)
    };
};
} // namespace DPL
```

All exceptions generated by a DPL object are derived from Base exception. Thus we can catch all specific object exceptions only.

To throw or catch DPL exceptions there are provided special macros:

Macro	Rule	Example

Try	Begin try block. Exact the same as standard try keyword.	Try
		{
		}

Catch	Catch an exception. It can be handled, ignored or rethrown in exception stack.	{
		}
		Catch(DPL::SqlConnection::Exception)
		{
		}

Throw	Throw an exception and break exception stack. Do not use in catch block (use ReThrow instead).	{
		Throw(MyException);
		}

© Samsung Electronics Co., Confidential

ThrowMsg	Throw an exception with message and break exception stack. Do not use in catch block (use ReThrow instead)	{
		ThrowMsg(MyException,"sample_error");
		}

ReThrow	Throw an exception in catch block. Cannot be used outside catch block.	Catch(Something)
		{
		ReThrow(EnclosingError);
		}

ReThrowMsg	Throw an exception with message in catch block. Cannot be used outside catch block.	Catch(Something)
		{
		ReThrowMsg(EnclosingError,"bad");
		}

CRYPTOGRAPHY

Cryptography module provided by DPL is a wrapper for OpenSSL routines giving easy access to hashing functions and cryptography algorithms.

HASH FUNCTIONS

A very easy wrapper for hashing functions is provided in DPL. There are more than 10 different hashing functions implemented in DPL (MD2, MD4, MD5, SHA, SHA1, DSS, DSS1, ECDSA, SHA224, SHA256, SHA384 and SHA512).

Example:

```
#include <dpl/crypto_hash.h>

DPL::Crypto::Hash::MD5 crypto;
crypto.Append("sample string");
crypto.Finish();
std::string hash = crypto.ToString();
```

Using hash algorithm is a simple as declaring a hashing function, appending some data with *Append()* and finally calling *Finish()* to calculate hash. Resulting string or raw data buffer can be obtained with *ToString()* method or *GetHash()*.

ALGORITHMS

In DPL there is provided a wrapper for blowfish encrypt/decrypt routine. Current interface works on *ISource* and *ITarget* interface. It will be deprecated and replaced by *AbstractInput/AbstractOutput* in future versions of DPL. One of possible usages of DPL cryptography algorithms is encrypting a file.

Example:

© Samsung Electronics Co., Confidential

```
#include <dpl/crypto_hash.h>

DPL::Crypto::Algorithm::FileSource source(“input.txt”);
DPL::Crypto::Algorithm::FileTarget target(“output.dat”);

DPL::Crypto::Hash key;
DPL::Crypto::Hash iv;

key.Append(“password”);
key.Finish();

iv.Append(“secret”);
iv.Finish();

DPL::Crypto::Algorithm::Blowfish algorithm(key.GetHash(), iv.GetHash());
Algorithm.Perform(&source, &target);
```

UTILITIES

DPL contains a lot of useful utilities that help developing robust and well designed MVC applications. To get familiar with each of them, the simplest way is start to use them.

ATOMIC

Atomic object is a low level primitive to do atomic reference counting. If guarantees that increment and decrement operations will be atomic. Increment operator returns no value. A boolean value indicating being still positive is returned after calling decrement operator. Internal value representation can be obtained with type definition *ValueType* from DPL::Atomic.

Atomic counters are used by shared pointer and shared array in DPL.

Example:

```
#include <dpl/singleton.h>

void Detach()
{
    […]
    if (!--m_sharedNode->ref)
    {
        delete m_sharedNode->ptr;
        delete m_sharedNode;
    }
    […]
}
```

SQL CONNECTION

© Samsung Electronics Co., Confidential

SQL Connection is a convenient wrapper for SQL connection. It is fully designed to cooperate with DPL internals. DPL adds support for Lucene indexer provided by platform. To enable it, select proper flag in constructor.

Note:

DPL SQL connection object does not implement any synchronization mechanism. All concurrent accesses to SQL will be directly passed to underlying SQL low-level objects. If any problems are to arise, they will be translated as a proper exception. Carefully use SQL connection object across different threads or provide some synchronization mechanism.

There are two types of command that are executed. One that returns no data and second that return *DataCommand* object which can be used to iterate through returned table. SQL parent object should not be deleted before all child *DataCommand* object are deleted. This will be checked by simple internal counting mechanism.

EXECUTING SIMPLE COMMANDS

To execute a simple SQL command without getting resulting table call:

Example:

```
#include <dpl/sql_connection.h>

DPL::SqlConnection connection;

void DeleteTables()
{
    connection.ExecCommand("DROP TABLE cars");
    connection.ExecCommand("DROP TABLE shops");
}
```

If any errors during execution of *ExecCommand* may arise, proper *SqlConnection::Exception* will be thrown.

EXECUTING COMMAND WITH RESULT

To execute full SQL command and retrieve result one must call *PrepareDataCommand*. Method returns an auto pointer, which can be saved and treated as a precompiled data command (prepared statement). Be aware to delete all precompiled data command before deleting *SqlConnection* object. Note that auto pointer has move policy, and *DataCommand* itself is noncopyable.

Example:

```
#include <dpl/sql_connection.h>
#include <string>
#include <list>

DPL::SqlConnection connection;
DPL::SqlConnection::DataCommandAutoPtr command;

std::list<std:string> GetUserNameListForID(int id)
{
    std::list<std::string> result;

    if (!command.get())
```


© Samsung Electronics Co., Confidential

```
        command = connection.PrepareDataCommand(
                    "SELECT name FROM users WHERE id=?");
    else
        command->Reset();

    command->BindInteger(1, id);

    while (command->Step())
            result.push_back(command->GetColumnString(0));

    return result;
}
```

If any errors during execution of PrepareData*Command* may arise, proper *SqlConnection::Exception* will be thrown.

Once precompiled command has been executed it must be *Reset()* before next usage. *Step()* method reads consecutive table row from result. If *Step()* returns false it indicates that no more rows are available.

There are several binding methods for statement. These are at least: *BindInteger()*. *BindString()*.

To retrieve column values from current row use methods: *GetColumnString()*, *GetColumnInteger()*, *GetColumnDouble()*.

SINGLETON

Singleton is one of most standard design pattern. We should use singleton where we have to ensure that only one instance of class exists. Examples include sound manager, core logic, IO controller, etc.

DPL's singleton implementation is a lazy singleton. It means that instance itself will not be allocated until first usage of it.

Example:

```
#include <dpl/singleton.h>

classGlobalClass
{
public:
    GlobalClass()
    {
    }
};

typedef Singleton<GlobalClass> GlobalClassSingleton;
```

To get instance of singleton, we call static method *Instance*. Instance method return reference to allocated instance:

Example:

```
GlobalClassSingleton::Instance().Foo();
```

To achieve best results, additionally we can make GlobalClass constructor private and give *friend* to the singleton class of GlobalClass.

© Samsung Electronics Co., Confidential

NONCOPYABLE

Noncopyable property is used to ensure that structure/class will never be copied. To make class or structure noncopyable privately derive from Noncopyable class.

Example:

```
#include <dpl/noncopyable.h>

class Collection : private Noncopyable
{
    std::list<int> integers;
    std::list<int>::const_iterator current;

public:
    Collection()
        : current(integers.begin())
    {
    }

};
```

RAII

RAII (Resource Acquisition Is Initialization) is a helper container that automatically manages resources and deletes it upon scope exit. There are several types of resources that can be automatically managed: memory allocated with *new*, memory allocated with *new []*, memory allocated with *malloc* or *calloc* and *file descriptor*.

Most popular is usage RAII objects in environment, where exceptions can occur. This is because, code automatically deletes all resources. Code is therefore cleaner and shorter.

Example:

```
#include <dpl/scoped_array.h>

ScopedArray<char> buffer(new char[data_size]);

if(read_input_data(buffer.Get(), data_size) == -1)
    throw ErrorReadInputData();

ScopedArray<char> parsed(new char[data_size]);

if(parse_buffer(buffer.Get(), parsed.Get()) == -1)
    throw ErrorParse();
```

SCOPED POINTER

Scoped pointer is one of simplest and most lightweight RAII mechanism. Use scoped pointer in all places, where fir some reason *dynamic allocation of memory* is needed and which needs to be *deleted after leaving scope*.

Example:

© Samsung Electronics Co., **Confidential**

```
#include <dpl/scoped_ptr.h>

for (file = files.begin(); file != files.end(); ++file)
{
    // for some reason we cannot make it on stack
    ScopedPtr<FileCleaner> cleaner(new FileCleaner());
    cleaner->Execute();
}
```

Do not use scoped pointer object for arrays. For arrays use scoped array object. Scoped pointer object is noncopyable.

SCOPED ARRAY

Scoped array is very similar mechanism to scoped pointer. The only difference is that it is intended to be used with arrays. It uses *delete []* operator instead of simple *delete*.

Example:

```
#include <dpl/scoped_ptr.h>

for (file = files.begin(); file != files.end(); ++file)
{
    // Assume that for some reason we cannot make it on stack
    ScopedArray<wchar_t> message(new wchar_t[file.size() + 1]);
    ConvertString(message.Get(), file.c_str());
    FooWide(message.Get());
}
```

Scoped array object is intended to be used with arrays only. Do not use for plain pointers. Scoped array object is noncopyable.

SHARED POINTER

Shared pointer is more advanced smart pointer that scoped pointer (and thus heavier mechanism). It uses *atomic reference counting*. It can be used in all places where object ownership is not clean or it is dynamically transferred between threads. After initialization of shared pointer with an object, it has reference count of one. Each copy of shared pointer, pointing the same object, increases atomically reference counter by one. Removing each referencing pointer decreases reference counter by one. If it drops to zero, pointer object is deleted.

Note:

Carefully design shared pointer dependencies. If circular dependencies are present, they can make deletion of pointer objects impossible. Use plain pointers or manual Reset where circular dependencies are present to break cycles.

Standard scenarios, where shared pointers are desired are event parameters. If event parameters are big structures or array, it is ineffective to copy it contents many times. Instead one can use shared pointer. Second scenario can be a return of newly created object from some factory. If someone forgets to save return value, and no smart pointer is used, it will result in memory leak. When shared pointer is used it will be successfully deleted, even if return value is not saved.

© Samsung Electronics Co., **Confidential**

Example:

```
#include <dpl/generic_event.h>
#include <dpl/shared_ptr.h>

class SomeClass {};
typedef DPL::SharedPtr<SomeClass> SomeClassPtr;

DECLARE_GENERIC_EVENT_1(SomeEvent, SomeClassPtr)

[…]

CONTROLLER_POST_EVENT(SomeController,
    SomeEvent(SomeClassPtr(new SomeClass("test", 123))));
```

Example:

```
#include <dpl/shared_ptr.h>

class Door {};

typedef DPL::SharePtr<Door> DoorPtr;

class CarPartGenerator
{
public:
    DoorPtr CreateDoor(int height)
    {
        Return DoorPtr(new Door(height));
    }
};
```

Shared pointer is intended to be used only with plain pointers. Do not use this class with pointers to arrays. For arrays use shared array object.

SHARED ARRAY

Shared array object is similar to share pointer object. The only difference is that it is intended to be used with arrays. Operator *delete []* is used. Do not use this object with plain pointers.

SCOPED FREE

Scoped free is similar to scoped pointer. The only difference is that it uses *free* instead of *delete operator*.

SCOPED CLOSE

Scoped close is a convenience object to call system *close* after scoped block is being left. This is useful when a file have to be opened and some operations done. During these operations an exception may be thrown. Scoped close ensures that in exception scenario it is properly closed (and thus all memory buffers are flushed to disk).

Example:

© Samsung Electronics Co., Confidential

```
#include <dpl/scoped_close.h>

char ReadByteFromFile(const char *filename)
{
    DPL::ScopedClose fd(open(filename, O_RDONLY));
    char byte;

    if (read(fd.Get(), &byte, sizeof(byte) <= 0)
      Throw(ReadError);

    return byte;
}
```

Scoped close object can also be used with all those system objects that upon destroy need to call *close* system call. They must also assume that -1 is an invalid descriptor that should be not closed.

SINGLE INSTANCE

Single instance is used to ensure that only one instance of application is running.

Example:

```
#include <dpl/single_instance.h>

SingleInstance g_singleInstance;

const char *SINGLE_INSTANCE_GUID =
               "327b0894-1234-6789-b15e-5c7d16514792";

if (g_singleInstance.TryLock(SINGLE_INSTANCE_GUID) == false)
{
    LogError("Already running!");
    return -1;
}

[…]

g_singleInstance.Release();
```

Single instance mechanism is a lightweight solution for application instance locking mechanism. Use this mechanism for locking applications instead of asynchronous semaphores.

Global application instance identifier should be carefully selected to be unique. GUID or similar naming is advised.

TASK / TASKLIST

Task and task list are patterns that make easy implementation of state machines. [***]

BINARY QUEUE

Binary queue is a very fast implementation of protocol accumulation buffer. It has two main purposes: adding some binary data to the end of buffer and consuming (removing) some data from its beginning.

© Samsung Electronics Co., Confidential

Implementation is based on linked list of data bucket. Each bucked can have different allocation scheme, especially custom data buffer. Methods provided by BinaryQueue are discussed in following table.

Method	Purpose	Assumptions
AppendCopy(**const void *buffer,** **size_t size** **)**	Appends a copy of memory from given pointer and given size	Pointer is not null, size can be zero
AppendUnmanaged(**const void *buffer,** **size_t bufferSize,** **BufferDeleter deleter =** **&BinaryQueue::BufferDeleterFree,** **void *userParam = NULL** **)**	Append an unmanaged portion of memory as a new bucket. Memory is deleted upon need with given Deleter. By default delete with *free()* is used. Do not free or reallocate unmanaged memory once appended to binary queue. Binary queue takes ownership of memory given.	Pointer is not null, size can be zero, deleter should delete all memory pointed by given pointer
AppendCopyFrom(**const BinaryQueue &other** **)**	Append a copy of data contained in another binary queue. A whole binary buffer is copied and appended to the end of target buffer. Implementation is efficient (moving pointers).	-
AppendMoveFrom(**BinaryQueue &other** **)**	Move all data from another binary queue to target binary queue. Implementation is efficient (moving pointers).	-
AppendCopyTo(**BinaryQueue &other** **)**	This is a convenience method similar to AppendCopyFrom, but in another direction.	-
AppendMoveTo(**BinaryQueue &other** **)**	This is a convenience method similar to AppenndMoveFrom, but in another direction.	-

© Samsung Electronics Co., Confidential

Size()	Return a total size of all data in bytes	-
Clear()	Delete all stored data in binary buffer	-
Empty()	Checks if binary queue is empty	-
Consume(size_t size)	Remove bytes from the beginning of binary queue (consuming data stream)	Size must be less or equal to binary queue size. A size of 0 bytes is a valid value.
Flatten(**void *buffer,** **size_t bufferSize** **)**	Copy bytes from binary queue to a raw data buffer. This automatically flattens bucket structure on fly.	Flattened byte count must be less or equal to binary queue size. Buffer cannot be null. A size of 0 bytes is a valid value.
FlattenConsume(**void *buffer,** **size_t bufferSize** **)**	This is a convenience method which is a sequence of flattening and consuming the same number of bytes from binary queue	Flattened byte count must be less or equal to binary queue size. Buffer cannot be null. A size of 0 bytes is a valid value.
VisitBuckets(**BucketVisitor *visitor** **)**	Visit internal bucket representation. This is a way of interpreting internal binary queue data without flattening. Used when even flatten operation is too costly.	Visitor cannot be null

ASYNCHRONOUS SEMAPHORE

Asynchronous semaphore is used for inter-process synchronization. Asynchronous semaphore can be used to create non-blocking mechanism for locking global named semaphores.

Only one type of event is generated from asynchronous semaphore. This is *AsyncLocked* event which is generated after lock has been taken. Be aware that this mechanism is heavier than single instance mechanism. For creating single instance exclusion for applications use *SingleInstance*, not *AsynchronousSemaphphore*. Semaphore mechanism is usually used for many consecutive locking accesses to some system resources (like a database file, configuration files or other).

© Samsung Electronics Co., Confidential

EVENT DELIVERY SYSTEM

For interprocess communication DPL provides a convenient mechanism of event delivery system. [***]

SAMPLE APPLICATIONS

A few real life sample applications will be provided to better see how DPL works. All samples and much more are provided in *examples* subdirectory in DPL repository. Reader is also encourages to look through unit-tests, because those more complicated usually have interesting DPL usages.

RPC METRONOME APPLICATION

An easy example will be a server which can be used as a metronome. The main component is a RPC sever. To connect to server client application can be used. After connection is established, server starts to send synchronization messages in exact 1 second intervals.

DPL classes that are used: *TcpSocketRPCServer, TcpSocketRPCClient*, *TcpSocketRPCConnection*, *Controller*.

Example:

```
//
// Metronome server
//
#include <dpl/tcp_socket_rpc_server.h>
#include <dpl/tcp_socket_rpc_connection.h>
#include <dpl/controller.h>
#include <dpl/application.h>
#include <dpl/log.h>
#include <algorithm>
#include <list>
#include <cassert>

// Metronome signal event
DECLARE_GENERIC_EVENT_0(SignalEvent)

// Heart beat interval
const double HEART_BEAT_INTERVAL = 1.0; // seconds

class MetronomeServerApplication
    : public DPL::Application,
      private DPL::Controller1<SignalEvent>,
      private DPL::EventListener<
                  DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent>,
      private DPL::EventListener<
                  DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent>,
      private DPL::EventListener<
              DPL::AbstractRPCConnectorEvents::ConnectionEstablishedEvent>
{
private:
    DPL::TcpSocketRPCServer m_rpcServer;

    typedef std::list<DPL::AbstractRPCConnection *> ConnectionList;
    ConnectionList m_connections;

    // Matronome signal received
    virtual void OnEventReceived(const SignalEvent &event)
```


© Samsung Electronics Co., Confidential

```
    {
        (void)event;

        // Signal all conection about heart beat
        DPL::RPCFunction proc;
        proc.AppendArg((int)0);

        for (ConnectionList::iterator it = m_connections.begin();
             it != m_connections.end(); ++it)
            (*it)->AsyncCall(proc);

        // Continue to emot heart beats
        DPL::ControllerEventHandler<SignalEvent>::
            PostTimedEvent(SignalEvent(), HEART_BEAT_INTERVAL);
    }

    void RemoveConnection(DPL::AbstractRPCConnection *connection)
    {
        // Find connection
        ConnectionList::iterator it = std::find(m_connections.begin(),
            m_connections.end(), connection);
        assert(it != m_connections.end());

        // Erase connection
        m_connections.erase(it);

        // Detach RPC connection listeners
        connection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent>::
                RemoveListener(this);
        connection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent>::
                RemoveListener(this);

        // Delete connection
        delete connection;
    }

    void AddConnection(DPL::AbstractRPCConnection *connection)
    {
        // Add connection
        m_connections.push_back(connection);

        // Attach event listeners
        connection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent>::
                AddListener(this);
        connection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent>::
                AddListener(this);
    }

    virtual void OnEventReceived(const DPL::AbstractRPCConnectionEvents::
                                     ConnectionClosedEvent &event)
    {
        (void)event;

        LogInfo("Connection closed");

        // Remove connection from list
```


© Samsung Electronics Co., Confidential

```
        RemoveConnection(
            static_cast<DPL::AbstractRPCConnection *>(event.GetSender()));
    }

    virtual void OnEventReceived(const DPL::AbstractRPCConnectionEvents::
                                 ConnectionBrokenEvent &event)
    {
        (void)event;

        LogInfo("Connection broken");

        // Remove connection from list
        RemoveConnection(
            static_cast<DPL::AbstractRPCConnection *>(event.GetSender()));
    }

    virtual void OnEventReceived(const DPL::AbstractRPCConnectorEvents::
                                 ConnectionEstablishedEvent &event)
    {
        // Save connection pointer
        LogInfo("New connection");

        // Add nre connection to list
        AddConnection(event.GetArg1());
    }

public:
    MetronomeServerApplication(int argc, char **argv)
        : Application(argc, argv, "rpc")
    {
        // Attach RPC server listeners
        m_rpcServer.DPL::EventSupport<
            DPL::AbstractRPCConnectorEvents::ConnectionEstablishedEvent>::
                AddListener(this);

        // Inherit calling context
        Touch();

        // Open RPC server
        m_rpcServer.Open(12345);

        // Start heart beat
        DPL::ControllerEventHandler<SignalEvent>::
            PostTimedEvent(SignalEvent(), HEART_BEAT_INTERVAL);

        // Started
        LogInfo("Metronome server started");
     }

    virtual ~MetronomeServerApplication()
    {
        // Delete all RPC connections
        while (!m_connections.empty())
            RemoveConnection(m_connections.front());

        // Close RPC server
        m_rpcServer.CloseAll();

        // Detach RPC server listener
```


© Samsung Electronics Co., Confidential

```
        m_rpcServer.DPL::EventSupport<
            DPL::AbstractRPCConnectorEvents::ConnectionEstablishedEvent>::
                RemoveListener(this);
    }
};

int main(int argc, char *argv[])
{
    return MetronomeServerApplication(argc, argv).Exec();
}
```

Example:

```
//
// Metronome client
//
#include <dpl/tcp_socket_rpc_client.h>
#include <dpl/tcp_socket_rpc_connection.h>
#include <dpl/application.h>
#include <dpl/log.h>

class MetronomeClientApplication
    : public DPL::Application,
      private DPL::EventListener<
          DPL::AbstractRPCConnectionEvents::AsyncCallEvent>,
      private DPL::EventListener<
          DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent>,
      private DPL::EventListener<
          DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent>,
      private DPL::EventListener<
          DPL::AbstractRPCConnectorEvents::ConnectionEstablishedEvent>
{
private:
    DPL::TcpSocketRPCClient m_rpcClient;
    DPL::ScopedPtr<DPL::AbstractRPCConnection> m_rpcConnection;

    virtual void OnEventReceived(const DPL::AbstractRPCConnectionEvents::
                                     AsyncCallEvent &event)
    {
        (void)event;

        // Heart beat
        LogInfo("* Got metronome signal *");
    }

    virtual void OnEventReceived(const DPL::AbstractRPCConnectionEvents::
                                     ConnectionClosedEvent &event)
    {
        (void)event;

        LogInfo("Connection closed");

        // Must quit
        Quit();
    }

    virtual void OnEventReceived(const DPL::AbstractRPCConnectionEvents::
                                     ConnectionBrokenEvent &event)
    {
        (void)event;
```


© Samsung Electronics Co., Confidential

```
        LogInfo("Connection broken");

        // Must quit
        Quit();
    }

    virtual void OnEventReceived(const DPL::AbstractRPCConnectorEvents::
                                 ConnectionEstablishedEvent &event)
    {
        // Save connection pointer
        LogInfo("Connected to metronome server");
        m_rpcConnection.Reset(event.GetArg1());

        // Attach event listeners
        m_rpcConnection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::AsyncCallEvent>::
                AddListener(this);
        m_rpcConnection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent>::
                AddListener(this);
        m_rpcConnection->DPL::EventSupport<
            DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent>::
                AddListener(this);
    }

public:
    MetronomeClientApplication(int argc, char **argv)
        : Application(argc, argv, "rpc")
    {
        // Attach RPC server listeners
        m_rpcClient.DPL::EventSupport<
            DPL::AbstractRPCConnectorEvents::ConnectionEstablishedEvent>::
                AddListener(this);

        // Open RPC server
        m_rpcClient.Open("127.0.0.1", 12345);

        // Started
        LogInfo("Metronome client started");
     }

    virtual ~MetronomeClientApplication()
    {
        // Delete all RPC connections
        if (m_rpcConnection.Get())
        {
            m_rpcConnection->DPL::EventSupport<
                DPL::AbstractRPCConnectionEvents::AsyncCallEvent>::
                    RemoveListener(this);
            m_rpcConnection->DPL::EventSupport<
                DPL::AbstractRPCConnectionEvents::ConnectionClosedEvent>::
                    RemoveListener(this);
            m_rpcConnection->DPL::EventSupport<
                DPL::AbstractRPCConnectionEvents::ConnectionBrokenEvent>::
                    RemoveListener(this);
            m_rpcConnection.Reset();
        }

        // Close RPC server
        m_rpcClient.CloseAll();
```


© Samsung Electronics Co., Confidential

```
        // Detach RPC server listener
        m_rpcClient.DPL::EventSupport<
            DPL::AbstractRPCConnectorEvents::ConnectionEstablishedEvent>::
                RemoveListener(this);
    }
};

int main(int argc, char *argv[])
{
    return MetronomeClientApplication(argc, argv).Exec();
}
```

Description:

Application is divided into two parts: a server and a client. Both are based on DPL application class. In server application, we server application class derivers from base application and additionally add support for listening for RPC connection events and RPC connector event (connected event). It is also a controller. One of possible simplest way of implementing exact metronome heart beats is to continuously send timed events to itself. If we stop to send next heart beat event, all process will be stopped. Heart beats are initiated in server application constructor by sending initial heart beat event.
Additionally, in server application constructor RPC server is initialized. Tcp RPC server (which can be replaced for example by Unix TCP server) is a type of RPC connector. It means that it generates connection established events. Such event contains new incoming connection (Abstract RPC connection). If desired, abstract RPC connection can be casted to a proper implementation of RPC connection (Unix/Tcp RPC connection), but usually it is not needed. Important note must be taken:

Note:

Always remember to attach *connection established* event handler. Incoming connections are dynamically allocated and resulting pointer is passed to event listeners. If there are no such, the pointer will be lost. Also remember to *delete all stored abstract RPC connections* (or use any of smart pointers).

When example server receives new connection, it attaches to all event supports from new connection. These are: *connection closed*, *connection broken* and *asynchronous call*.

If connection closed or connection broken event is received, from event sender field, we extract which abstract RPC connection was closed or broken. Then all event listeners are removed, stored connection removed and finally object *is deleted*.
Asynchronous calls from client are simply ignored. They are not used in this example server.
If a heart beat event is generated and handled, all stored connections are enumerated, and for each of them, a RPC call is executed.
Note that all these operations are in the same context – main application loop. There is no need to add any synchronization mechanism for stored connections list.

Client application is even simpler. Similarly to RPC server, a RPC client is a type of RPC connector. It generated connection established events, for which we should register a listener. To initiate a connection, *Open()* method is used. Some example parameters are given. After connection is established, we store the only connection to server, and register for all RPC connection events. Connection closed and connection broken event are handled in a way that application simply quits. Asynchronous calls are interpreted here. These are heart beats from server. After such is received, simple message is generated to screen.

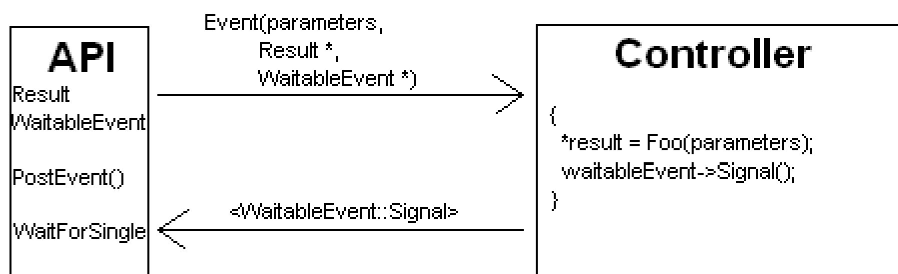
© Samsung Electronics Co., **Confidential**

SYNCHRONOUS EVENT CALLS

Implementing synchronous event calls, executed in dedicated controller thread. This kind of situation can occur when a synchronous API is implemented on top of asynchronous API. This is a standard situation; application core logic is usually asynchronous.

To effectively solve this kind of synchronization, WaitableEvent object can be used. Scenario is as follows: an event with parameters for call, pointer to resulting structure and a pointer to a waitable event is posted to a controller. Controller is working in context of dedicated thread. Upon receiving event, it processes it and sets resulting structure pointed by pointer passed in event. After all operations are done, controller signals WaitableEvent. From the other side, we post an event, and simply wait until waitable event is signaled.

Diagram:



Example:

```
#include <dpl/controller.h>
#include <dpl/generic_event.h>
#include <dpl/waitable_event.h>

DECLARE_GENERIC_EVENT_3(AsyncEvent, int, int *, WaitableEvent *)

void SynchronousAPICall(int parameter)
{
    WaitableEvent syncEvent;
    int result;
    int param = 23;

    CONTROLLER_POST_EVENT(FooController,
        AsyncEvent(param, &result, &syncEvent);

    DPL::WaitForSingleHandle(syncEvent.GetHandle());

    printf("result is: %i\n", result);
}

[…]

FooController::OnEventReceived(const AsyncEvent &event)
{
    int param = event.GetArg0();
    int *result = event.GetArg1();
    DPL::WaitableEvent *syncEvent = event.GetArg2();

    *result = param >> 1;
```


© Samsung Electronics Co., **Confidential**

```
    syncEvent->Signal();
}
```


© Samsung Electronics Co., **Confidential**